Rapport final NoSQL

TRINQUART Matthieu, SANCHEZ Martin

1 Introduction

1.1 Structure du projet

Notre projet contient plusieurs dossier

- qengine : Il s'agit de notre système d'évaluation de requêtes en étoile en Java.
- JenaTester : Nous avons utilisé Jena pour évaluer les requêtes sélectionnés pour nos tests. Ainsi que deux scripts python pour les trier.
- Benchmark : Il contient les test ainsi que leurs résultats.
- watdiv-mini-projet : Il s'agit du programme fournit en cour pour générer les donnés. (Il ne se trouvera pas dans le rendu car nous ne l'avons pas modifié)

1.2 Le projet

Dans ce projet nous avons dans un premier temps réalisé un système d'évaluation de requêtes en étoile.

Notre système d'évaluation se base sur le format RDF, les données sont donc stockés sous forme de triplés Sujet Prédicat Objet (SPO).

Ce schéma montre ce à quoi ressemble une requête en étoile :

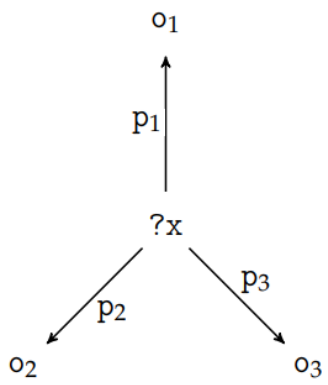

FIGURE 1 – Description d'une requête en étoile. (x étant le Sujet)

Afin de traiter les requêtes, nous avons mis en place certains concepts, comme un Dictionnaire, pour attribuer à chaque mot une valeur numérique, cela permet de raisonner avec des entier ce qui est plus simple pour une machine. Nous avons aussi implémenté un index, plus précisément 6, pour organiser les données et facilité leur accès.

1.3 Tester l'efficacité de notre solution

La seconde partie de ce projet consistait à évaluer les performances de notre programme. Pour cela nous avons ajouter a notre projet un certain nombre d'options pour qu'il puisse être utilisé en lignes de commande.

— queries : le fichier de requêtes
— data : le fichier de donnés
— output : le fichier de sortie qui contiendra les temps d'exécutions
— Jena : permet d'activer la vérification avec Jena
— warm : permet d'échauffer le système avec un certain nombre de requêtes
— shuffle : permet de mélanger la liste de requêtes avant l'exécution

Vous pouvez trouver le code source dans le fichier "qengine".

2 Préparation du protocole de test

Pour cette partie du projet, nous avons utilisé WatDiv pour générer plusieurs jeux de donnés et plusieurs jeux de requêtes. Ce qui nous permettra de tester notre programme dans des cas d'utilisation très variés. Nous avons donc généré des jeux de donnés allant de 500 000 triplés à 2 000 000.

2.1 Jeux de requêtes

Pour réaliser nos test il est important de choisir des requêtes pertinentes. Pour cela nous en avons généré des grandes quantités afin de pouvoir les sélectionner avec soin.

Nous avons décidé d'imposer des quotas de requêtes à 0 résultats, 1, 2, 3 et 4 conditions. Cependant nous avons eu du mal à trouver assez de requêtes à 4 conditions donnant des résultats, c'est pourquoi nous avons revu ces quotas à la baisse.

Nous obtenons alors un jeu de 3000 requêtes sans doublons.

Vous pouvez trouver tout le code source dans le fichier "JenaTester".

2.2 Hardware et Software

Pour nos test nous disposont du matériel suivant :
Processeur :

- Vendor : AMD
- Model : Ryzen 5 4600H with Radeon Graphics
- Cores : 6 Cores, 12 Threads
- Clockspeed : 3.0GHz
- L2 Cache 3MB
- L3 Cache 8MB

Mémoire vive :

- Size : 8GB RAM

Disque :

- Model : KINGSTON OM8PCP3512F-AB
- Size : 476.9 GB
- Sequential Read : 1,358 MBytes/Sec
- Sequential Write : 666 MBytes/Sec

Software :

- OS : PopOs
- Version : 22.04 LTS

Nous allons nous intéresser principalement aux temps d'exécutions des programmes pour les comparer entre eux.

3 Les tests

3.1 Structuration des fichiers

Pour le bon déroulement des test, le fichier contient les jar des différents projets : ”qengine” et ”JenaTester” dans le fichier Jars, dans les fichier ”data” et ”queries” vous trouverez les donnés et le requêtes. Et dans le fichier ”ressources” certains éléments comme la sérialisations sur disque des Dictionnaire et Index.

Nous avons un script bash par test à effectuer. ”init.sh” permet d'initialiser un certain nombre de variables et de fonctions qui nous servent dans tout les tests. Les séparer dans plusieurs fichier permet de séparer chaque exécution, nous avons pensé que laisser un temps entre chaque test permettrait d'obtenir un environnement de test plus table.

Il y a d'autre fichier .sh qui permettent de faire les test suivant :

— test_coldVSwarm.sh : Teste le systeme en faisant varier le -warm.
— test_dataSize.sh : Teste le système en faisant varier la taille des donnés.
— test_equalToJena.sh : Teste si notre système rends les mêmes résultats que Jena.
— test_qEnginevsJena.sh : Compare les temps d'execution de notre système avec ceux de Jena.
— test_ramMatter.sh : Compare les temps d'execution selon la ram allouée.

Vous pouvez trouver tout le code source dans le fichier ”Benchmark”.

3.2 Réalisation des tests

Est-ce que notre programme donnée toujours les bon résultats ?

Difficile a démontrer par le calcul, cependant nous pouvons exécuter notre programme avec tous les jeux de données et toutes les requêtes dont nous disposons tout en activant la vérification avec Jena.
Voila les résultats :

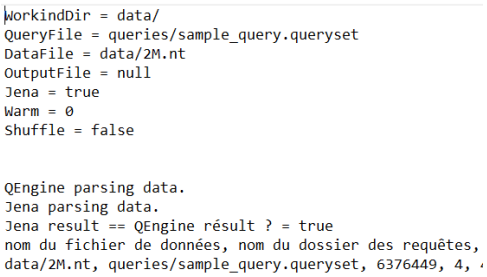

FIGURE 2 – Exemple d'exécution en activant la validation par Jena. (-Jena)

Jena result == Qengine résult ? = true nous signale que notre programme n'as trouvé aucune différence entre les résultats de Jena et les nôtres.

Est-ce que les temps d'exécutions varient si le système est chaud ?

Pour vérifier cela, nous avons lancé un certain nombre de fois notre programme, dans un premier temps sans utiliser l'option warm, puis en échauffant notre système avec 200, 500 et 1000 requêtes.
Voici ce que nous avons obtenu : (Les images sont disponibles dans le rendu)

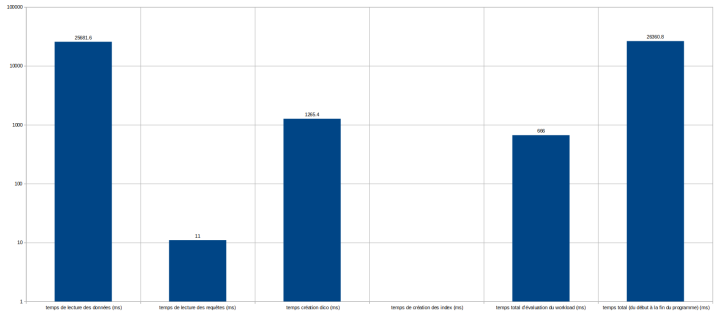

FIGURE 3 – Temps d'exécution sans échauffement.

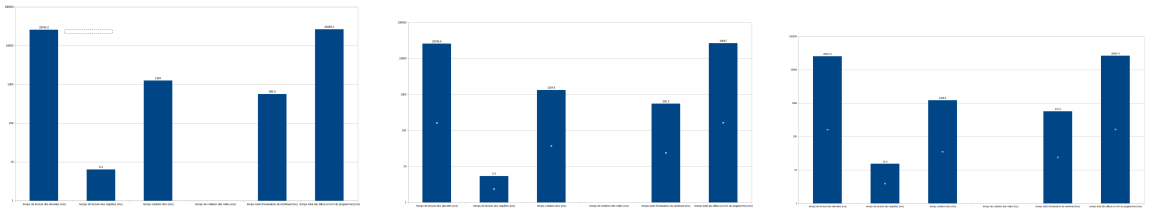

FIGURE 4 – Temps d'exécution avec 200, 500 et 1000 requêtes d'échauffement.

Le résultat nous montre une légère baisse du temps d'exécution du jeu de requêtes, cette baisse n'est pas particulièrement élevée de l'ordre de 100ms, cependant nous pouvons noter qu'en échauffant notre système nous augmentons le nombre de requêtes à exécuter.

Sommes nous plus rapide que Jena ?

Nous allons tester le temps d'exécution de notre programme ainsi que de Jena. Pour cela nous avons exécuté chaque Jar 10 fois. Nous comparons les résultats dans ce diagramme :

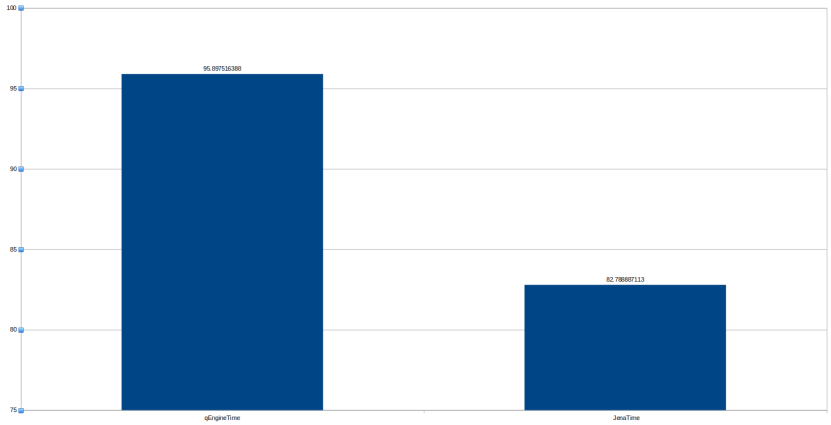

FIGURE 5 – Comparaison des temps d'exécution totaux.

Nous remarquons que malgré le fait que nous ne traitions qu'un type de requêtes, Jena est plus rapide que notre programme. Cela est du au temps de création du Dictionnaire qui représente dans chaque exécution environ 80% du temps d'exécution total.

Est-ce que plus de mémoire implique plus de performances ?

Pour ce test nous avons voulu vérifier si la mémoire vive de la machine virtuelle Java pouvait impacter les performances, nous l'avons donc fait varier selon le matériel que nous possédions.

Ainsi sur 10 exécutions, nous obtenons les résultats suivants :

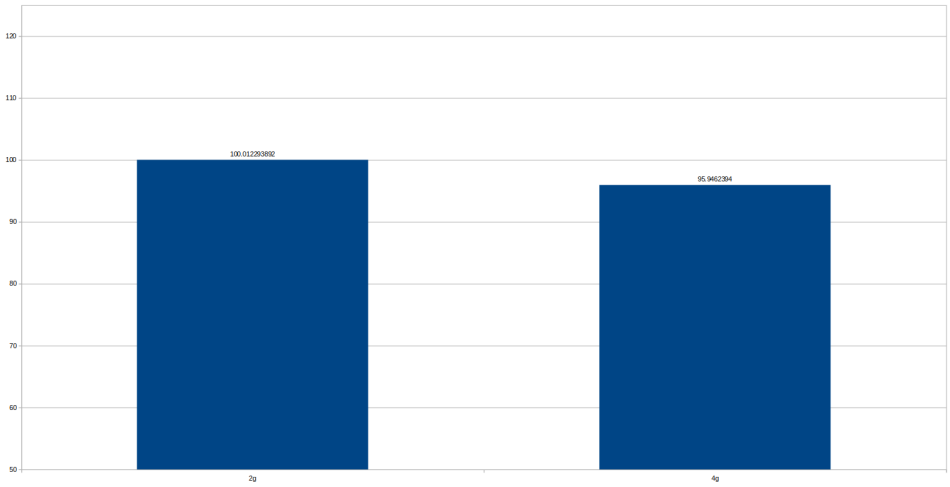

FIGURE 6 – Comparaison des temps d'exécution selon la quantité de mémoire vive alloué.

On remarque une légère baisse du temps d'exécution, même si cela n'est pas énorme, nous pouvons penser que la quantité de mémoire vive impacte le temps d'exécution. Pour vérifier notre hypothèse, nous pourrions exécuter ce test sur une autre machine possédant plus de mémoire vive.

Est-ce que plus de donnés implique des temps d'exécution plus long ?

Étant donné que dans notre cas, ce qui prends le plus de temps est la création du Dictionnaire, nous nous sommes demandé si l'augmentation des données faisait augmenter significativement le temps d'exécution.

Ainsi sur 10 exécutions, nous obtenons les résultats suivants :

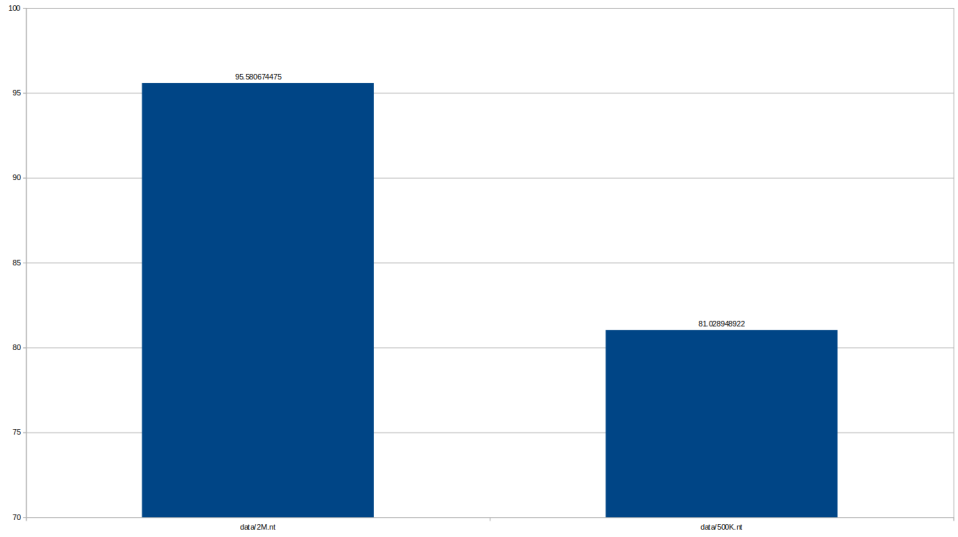

FIGURE 7 – Comparaison des temps d'exécution selon la quantité de donnés.

Nous voyons que l'augmentation des donnés fait augmenter le temps d'exécution, ce qui n'est pas surprenant, cependant, l'augmentation n'est que de 20% en ayant multiplié par 4 la quantité de donné.